Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 258 (2009) 3–20

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Graphical User Interface for Maude-NPA

S. Santiago[a,2] C. Talcott[b,3] S. Escobar[a,4] C. Meadows[c,5] J. Meseguer[d,6]

[a] *Universidad Politécnica de Valencia, Valencia, Spain*

[b] *SRI International, Menlo Park, USA*

[c] *Naval Research Laboratory, Washington DC, USA*

[d] *University of Illinois at Urbana-Champaign, Urbana, USA*

Abstract

This paper presents a graphical user interface (GUI) for the Maude-NPA, a crypto protocol analysis tool that takes into account algebraic properties of cryptosystems not supported by other tools, such as cancellation of encryption and decryption, Abelian groups (including exclusive or), and modular exponentiation. Maude-NPA has a theoretical basis in rewriting logic, unification and narrowing, and performs backwards search from a final attack state to determine whether or not it is reachable from an initial state. The GUI animates the Maude-NPA verification process, displaying the complete search tree and allowing users to display graphical representations of final and intermediate nodes of the search tree. One of the most interesting points of this work is that our GUI has been developed using the framework for declarative graphical interaction associated to Maude that includes IOP, IMaude and JLambda. This framework facilitates the interaction and the interoperation between formal reasoning tools (Maude-NPA in our case) and allows Maude to communicate easily with other tools.

Keywords: Graphical user interface, Maude-NPA, IOP, IMaude, JLambda, Maude

[1] S. Santiago and S. Escobar have been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2007-68093-C02-02, and Generalitat Valenciana under grant GVPRE/2008/113. Carolyn Talcott is supported by the NSF under grant IIS-0513857.

[2] Email: ssantiago@dsic.upv.es

[3] Email: clt@csl.sri.com

[4] Email: sescobar@dsic.upv.es

[5] Email: meadows@itd.nrl.navy.mil

[6] Email: meseguer@cs.uiuc.edu

1571-0661 Published by Elsevier B.V.

doi:10.1016/j.entcs.2009.12.002

1 Introduction

The Maude-NRL Protocol Analyzer is a tool for analyzing cryptographic protocols that takes into account the equational properties of the cryptosystems involved. These include, for example, exclusive or and Diffie-Hellman exponentiation. But a key property of Maude-NPA that it is designed to, is be *extensible*, that is, the procedures and techniques used by the tool are general enough so that they can be readily extended to include new equational theories as they arise.

A good graphical user interface can do much to help the extensibility and usability of Maude-NPA. For the general user, it can provide graphical representations of both successful and failed attack paths, helping the user to understand the reasons behind the success or failure of a protocol. For an advanced user, who may be trying out new equational theories, it can provide a closer look at the search tree, indicating where a state explosion is occurring that may indicate a problem that must be redressed by better mechanisms for the new theory. For the tool developers, a close look at problematic nodes on both full and partial search trees may gain insight into why the state explosion is occurring and what mechanisms need to be devised or improved to address the problem. But, in order to provide all these services, the GUI must animate, not only successful attacks, but the entire Maude-NPA search process. Thus, the services it provides must go much beyond those offered by a usual GUI.

The desirable features mentioned above for the Maude-NPA GUI are not obtained by following a standard approach to GUI development. Indeed, we seriously doubt that a standard approach using a conventional programming language could have given us the simplicity, levels of abstraction, and flexibility for the GUI features and ease of evolution that we need in a tool like the Maude-NPA whose design is still evolving. Instead, our approach to GUI design is based on a philosophical viewpoint that regards GUI interactions as *another form of rewriting*; specifically, it is based on a *declarative, rewriting-based approach to GUI design.*

We can summarize this declarative approach as follows. First of all, the rewriting logic approach to concurrent object interaction [20,21], in which object transitions are specified by rewrite rules, is adopted as the semantic framework for *both* the Maude-NPA, where cryptographic protocols are indeed specified this way, and for the Maude-NPA's GUI. That is, the tool's GUI and its interactions with the user are also specified by rewrite rules in which proxy GUI objects interact with the user and with an encapsulation of Maude-NPA by message passing. Second, a GUI implementation is then *derived from its rewriting logic specification* by mapping these proxy objects

to corresponding graphical *objects* that appear like *built-in external* objects supported by the Maude infrastructure [8]. The implementation of the graphical objects and message passing communication with these external objects is achieved using the InterOperability Platform (IOP), that supports the actor model, and its associated IMaude library [18] (see Section 3). The infrastructure used to build the Maude-NPA GUI is quite general, extending Maude to provide a framework on top of which many interesting applications and graphical interfaces can be built.

1.1 Related work

The area of formal analysis of cryptographic protocols has been an active one since the mid 1980's. The idea is to verify protocols that use encryption to guarantee secrecy and that use authentication of data to ensure security, against an attacker (commonly called the *Dolev-Yao* attacker) who has complete control of the network, and can intercept, alter, and redirect traffic, create new traffic on his/her own, perform all operations available to legitimate participants, and may have access to some subset of the longterm keys of legitimate principals. In the simplest case, cryptosystems are assumed to behave like black boxes: an attacker knows nothing about encrypted data unless it has the appropriate key. In more sophisticated analyses, the cryptosystem may be assumed to obey a set of equational properties, as in our own tool, Maude-NPA, or may even be fully specified in the type of computational model used by cryptographers. Whatever approach is taken, the use of formal methods has had a long history, not only for providing formal proofs of security, but also for uncovering bugs and security flaws that in some cases had remained unknown long after the original protocol's publication.

A number of approaches have been taken to the formal verification of cryptographic protocols. One of the most popular is model checking, in which the interaction of the protocol with the attacker is symbolically executed. Indeed, model-checking of secrecy (and later, authentication) in protocols in the bounded-session model (where a *session* is a single execution of a process representing an honest principal) has been shown to be decidable [24], and a number of bounded-session model checkers exist. Moreover, a number of unbounded model checkers, of which Maude-NPA is one, either make use of abstraction to enforce decidability, or allow for the possibility of non-termination.

The earliest tools, such as the Interrogator [16] and the NRL Protocol Analyzer (NPA) [19], while not strictly speaking model checkers, relied on state exploration, and, in the case of NPA, could be used to verify security properties specified in a temporal logic language. Later, researchers used generic model checkers to analyze protocols, such as FDR [17] and later Murphi [22].

More recently the focus has been on special-purpose model checkers developed specifically for cryptographic protocol analysis, such as Blanchet's ProVerif [6], the AVISPA tool [4], and Maude-NPA itself [13].

One of the advantages of using model-checkers is that they give explicit counter-examples, which can provide insights into the reasons why a system fails to satisfy its specification. In the case of cryptographic protocol analysis tools, such counterexamples are very amenable to graphical expression: each process (including attacker processes) can be written as a sequence of nodes, which represent either sending or receiving messages. If a process sends a message to another, this can be written as an arrow from the sending node to the receiving node. The popular strand space model [27], upon which a number of tools, including Maude-NPA, is based, makes use of such notation and formalizes it.

GUI interfaces for protocol analysis tools have thus traditionally focused on model-checkers. They generally fall into two categories: interfaces that assist the user in specifying protocols that are input into the tool, and interfaces that assist the user in understanding the output.

Probably the earliest case of the latter was one of the first protocol analysis tools developed, the Interrogator [16]. The Interrogator performed a depth-first backwards search from an insecure state. Each time it could go no further along a path, whether the result was successful or not, it would display the result in a graphical form, thus giving an animation of the tool's search process. A graphical representation of a normal execution was also used to assist the user to specify attack states to be searched for.

A more recent example of a graphical output interface is the Scyther tool [9]. It searches from instances of roles, generating all trace patterns that are consistent with it. The Scyther tool outputs graphical representations of these trace patterns. Unlike the Interrogator, however, it does not output partial or unsuccessful patterns.

Regarding GUIs that assist protocol specifications, one prominent example is SPAN [7], a companion tool for AVISPA. SPAN is a graphical protocol animator that allows a user to check the validity of specifications written in the AVISPA input language, HLPSL. SPAN allows the user to step through executions of a protocol in the absence of an intruder, determining whether or not it is behaving according to the intended way. SPAN can also be used with the intruder present to construct attacks on protocols, although it does not find them automatically; the user must choose each step to execute out of a set presented to him by the tool.

Another example of a GUI that assists protocol specifications, with a very different approach, is the Protocol Derivation Assistant (PDA) [3]. PDA al-

lows the user to specify not only single protocols, but families of protocols, using graphical techniques for constructing new protocols out of old ones. PDA also supports integration with different specification and analysis frameworks, so it can be used as an interface with different formal tools.

All of the above GUIs, except to a certain extend the Interrogator, have one thing in common. They assist the user either in specifying input into a protocol analysis tool or interpreting the output of the tool. But they do not provide much insight into the workings of the analysis tool itself. But this can be very helpful, especially in a tool like Maude-NPA that takes an extensible approach to incorporating new equational theories. Being able to examine the results of partial and failed searches can help a user understand why a particular protocol or a new equational theory is producing a state explosion or infinite loops, and what needs to be done to address the problems. It can also provide assistance in the debugging of protocol specifications, in a way similar to the SPAN tool. Finally, an understanding of partial and failed searches can be useful in identifying the key assumptions upon which a protocol's security depends, a piece of information which is otherwise difficult to get from model-checkers. For these reasons we have decided to develop an interface for Maude-NPA that goes beyond what GUIs for protocol analysis tools usually provide, and gives the user a complete animation of the Maude-NPA search tree generation process which she can search at will.

1.2 Structure of the paper

The rest of this article is organized as follows. In Section 2, we introduce the Maude-NPA tool and its use. Section 3 presents some background about the IOP, IMaude and JLambda frameworks. In Section 4, we give some details on how the Maude-NPA GUI has been actually implemented using the IOP-IMaude framework. Section 5 describes in detail the features of the Maude-NPA GUI, including some pictures. In Section 6, we finish with some conclusions and plans for future work.

2 Maude-NPA

Maude-NPA is a tool for finding attacks, or proving their absence, in a cryptographic protocol. It uses backwards search from insecure states. It analyzes infinite state systems including an active intruder, making no abstraction or approximation of nonces, and with an unbounded number of sessions. In order to reduce the search space it uses various optimization techniques, to prevent infinite loops and to avoid useless transitions to unreachable states. The tool

is publicly available [7], including a user manual and some protocol examples.

One major feature of the tool is that it supports algebraic identities obeyed by the crypto-algorithm, such as exclusive or, exponentation and encryption/decryption cancellation. The ultimate goal, however, is for the equational theories offered by Maude-NPA to be *user extensible* so that new equational theories falling into a broad class can be introduced.

The protocol model used by Maude-NPA is based on the popular strand space model introduced by Thayer, Herzog and Guttman in [27]. Each local execution or session of an honest principal is represented by a sequence of positive and negative terms called a *strand*. These terms are built from variables, function symbols and constants. *Negative terms* represent received messages, and *positive terms* stand for sent messages. An example of a strand is given below:

$$[\ pke(B, N_A; A)^+,\ pke(A, N_A; N_B)^-,\ pke(B, N_B)^+\ ]$$

This strand tells us that a principal A first sends a message with a nonce (N_A) concatenated with her name (A) encrypted under B's public key. Then, she receives a message encrypted with her public key that contains her previously sent nonce concatenated with a nonce (N_B) (presumably from B). Finally, she sends the nonce N_B encrypted under B's public key.

Each intruder action is also represented by a strand. The following intruder strand

$$[\ X^-,\ Y^-,\ (X; Y)^+\ ]$$

shows that the intruder has the ability to concatenate two messages, once he or she has learned them.

A strand may contain variables, except for terms of type `Fresh`, which are always treated as constant (i.e., they are used in nonces). Strands are annotated with the variables of type `Fresh` generated by that principal. The following example is an honest principal strand specified in Maude-NPA and annotated with the fresh variable "r" used for nonce $n(A, r)$:

$$::\ r\ ::\ [\ pke(B, n(A, r); A)^+,\ pke(A, n(A, r); N_B)^-,\ pke(B, N_B)^+\ ]$$

As one of the techniques to avoid infinite loops, Maude-NPA relies on the assumption that an attacker never learns a term more than once, so the tool must keep track of what terms are learned and *when* during a backwards search. Backwards search means that we know what the intruder knows in the future, but we have an incomplete picture of what the intruder learned in the past. We need to determine the concrete moment when the intruder learns something. Such augmentations of the strand space model to include state

[7] At http://maude.cs.uiuc.edu/tools/Maude-NPA/

are not new (see for example [23], which includes an explicit representation of the knowledge of each principal), but the Maude-NPA requirements are somewhat unusual in that the use of backwards search necessitates an explicit representation of what will happen in the future. Thus the original strand space model is augmented to include the notion of time in terms of "past", "future", and "present". The strands used in Maude-NPA include a marker which denotes the current state of the execution. This marker separates the past and the future messages of a strand and allows Maude-NPA to perform a backwards search through the strands from a final state towards an initial state. The notion of present tells what the intruder knows in the present, which is denoted by expresions of the form $t \in I$. The notion of future means what the intruder does not know in the present but will definitely learn in the future, which is denoted by expresions of the form $t \notin I$.

In our tool a *state* is represented by a set of strands plus the intruder knowledge and looks like this:

$$[\ m_{11}^{\pm},\ \ldots \mid \ldots,\ m_{k_1 1}^{\pm}\ ]\ \&\ \ldots\ \&\ [\ m_{1n}^{\pm},\ \ldots \mid \ldots,\ m_{k_n n}^{\pm}\ ]$$
$$\&\{t_1 \notin I, \ldots, t_m \notin I, s_1 \in I, \ldots, s_{m'} \in I\}$$

where:

(i) Each strand is divided into past and future $[\ m_1^{\pm}, \ldots, m_i^{\pm} \mid m_{i+1}^{\pm}, \ldots, m_k^{\pm}\ ]$, where $m_1^{\pm}, \ldots, m_i^{\pm}$ is the past, $m_{i+1}^{\pm}$ is the present, and $m_{i+2}^{\pm} \ldots, m_k^{\pm}$ is the future.

(ii) An initial strand is one in which the past part is empty: $[\ nil \mid m_1^{\pm}, \ldots, m_k^{\pm}\ ]$. On the other hand, a final strand is one in which the future part is empty: $[\ m_1^{\pm}, \ldots, m_k^{\pm} \mid nil\ ]$.

(iii) The intruder knowledge contains terms of the form $m \in I$ and $m \notin I$, where $m \in I$ means what the intruder knows m in the present and $m \notin I$ denotes terms the intruder does not know in the present but will definitely learn in the future.

(iv) The initial intruder knowledge is a set without $m \in I$ terms, i.e., a set of the form $\{t_1 \notin I, \ldots, t_n \notin I\}$, whereas the final intruder knowledge is a set without $m \notin I$ terms, i.e., a set of the form $\{s_1 \in I, \ldots, s_m \in I\}$.

From a given final attack state, Maude-NPA executes the protocol backwards to an initial state, if possible. The first step before actually starting the backwards search is to prove lemmas about unreachable states by generating *the grammars of the protocol*. These grammars help to reduce the search space (see [11]) and are automatically generated by the tool. For each intermediate state found in the backwards execution, the tool checks whether it is

unreachable using such lemmas (and other techniques, see [12]) and discards it if so.

The analysis must start with a final state describing the attack state and including some final strands and some intruder knowledge, typically with terms of the form $m \in I$. Variables in the final attack state are appropiately instantiated by backwards narrowing. The number of strands and intruder facts is increased by backwards narrowing.

Let us briefly explain how the backwards narrowing analysis is performed. Given a state, we fix a concrete strand in the state and a message $m_{ij}^{\pm}$ in the past part of such strand. If $m_{ij}^{\pm}$ is a negative node (m_{ij}^{-}), then it is (i) either E-unified with a term s_k already known by the intruder (i.e., a fact $s_k \in I$ appears in the intruder knowledge of the given state), or (ii) included into the intruder knowledge of the given state as a new challenge $m_{ij} \in I$, but only if it is not already present in the intruder knowledge. Action (i) means that the message m_{ij}^{-} received by a strand comes from the intruder, who knows that message. Action (ii) means that the intruder must know message m_{ij} and we include it into the set of challenges for the intruder. If $m_{ij}^{\pm}$ is a positive node (m_{ij}^{+}), it is (iii) either accepted but without increasing the intruder knowledge, or (iv) E-unified with a term s_k known by the intruder (i.e., a fact $s_k \in I$ appears in the intruder knowledge of the given state), and the fact $s_k \in I$ is transformed into $s_k \notin I$. Action (iii) means that the message m_{ij} output by a strand is not relevant for the intruder. Action (iv) means that the message m_{ij} output by a strand is used by the intruder to learn it, actually describing when he learned it. In order to allow an unbounded number of strands, the tool adds new strands containing messages of the form m_j^{+} such that m_j E-unifies with a message s_k known by the intruder (i.e., $s_k \in I$ appears in the intruder knowledge) and also transforms the fact $s_k \in I$ into $s_k \notin I$. Further details and protocol examples can be found in [13].

3 The InterOperability Platform (IOP) and IMaude

The InterOperability Platform (IOP) [18] was developed to facilitate the interaction and interoperation between formal reasoning tools. In particular IOP enables Maude to communicate with other tools, thus making it a good starting point for developing a GUI for Maude-NPA. IMaude is a set of Maude modules that provide basic features needed to program interactive assistants in Maude. The combination of IOP and IMaude has been used in several projects, including Remote Agents (providing an interactive graphical interface to an executable specification of a simple remote robot) [10] and the Pathway Logic Assistant [26,25] (an interactive graphical interface for visual-

izing and analyzing formal models of cellular signaling processes).

The next subsections explain in more detail the main components used to develop the Maude-NPA GUI: the IOP Interaction Model, Interactive Maude (IMaude), and the graphical interaction package g2d.

3.1 IOP Interaction Model

The IOP Interaction model is based on the actor model of distributed computation [14,15,5,1,2]. The actor model is a model of distributed computation based on the notion of independent computational agents, called actors, that interact solely via message passing. An actor can create other actors; send and receive messages; and modify its own local state. An actor can only affect the local state of other actors by sending them messages, and it can only send messages to its acquaintances—either actors whose names it was given upon creation, or names it received in a message or names of actors it created. Actor semantics admits only fair computations, which in the simplest case means reliable message delivery.

In the IOP architecture there is a dynamic pool of actors, since new actors can be created and existing actors can be destroyed. There are also three independent processes that manage these actors: (i) *Main*, which creates and configures the system; (ii) *Registry*, also called the system actor, that keeps track of the current actors and maintains the lines of communications; and (iii) *IOP GUI.editor*, which allows the user to communicate with any of the actors [8].

An IOP actor is one or more UNIX style processes registered within the system. Each one has three FIFOs (i.e., UNIX style pipes). Its `stdin`, `stdout` and `stderr` file descriptors are redirected to one of these FIFOs. An actor can be created at the startup (these are the initials actors), in response to some event, or by the user or another process asking the system actor to do so.

IOP comes with a basic set of actors. The most important for our purposes are: (i) the System actor, (ii) the GUI.editor, (iii) the Maude actor, and (iv) the Graphics2D actor. The Maude actor is a Maude process encapsulated in a wrapper that enables interaction with other actors.

Communication between actors is performed via asynchronous message passing through the IOP registry, which acts as a post office, routing the messages from each sender actor to the desired target actor. There exist several forms of communication: (i) *inter-actor*, i.e., from one actor to another; (ii) *meta-actor*, i.e., from an actor to the registry; and (iii) *interface*, i.e., between the GUI.editor and an actor (the registry or any other actor).

[8] This should not be confused with the Maude-NPA GUI, since they are not the same.

3.2 Interactive Maude (IMaude)

Interactive Maude (IMaude) is a collection of Maude modules that support writing interactive Maude applications where rewriting is interleaved with communications with the environment and the IMaude application's state persists across communications.

An IMaude state has the form

```
[ input, st( control, wait4s, requests, environment, log),
                                output ]
```

where the notation "`[input,state,output]`" is standard for user interfaces in Maude (see `LOOP-MODE` in [8]) and each IMaude component is described as follows:

- *control:* it contains either the request currently being processed or the constant `ready` if there is none.
- *wait4s:* a set of the suspended tasks waiting to handle incoming messages.
- *requests:* a queue of requests waiting to be processed by IMaude.
- *environment:* contains a set of entries mapping identifiers to data values.
- *component:* a list of log items for debugging by allowing events and status to be recorded as requests are being processed.

IMaude provides rules for receiving, queueing, and scheduling requests (generated by incoming IOP messages, or by internal rewrites), defining continuations to receive replies to outgoing IOP messages. It provides mechanisms to ensure sequential processing when needed, and enabling reactive response. Built-in requests include saving and restoring the state, and access to the file system.

3.3 The g2d package and the Graphics2d actor

The *graphical interaction package* g2d is a Java package designed to simplify developing interactive visualizations for formal models and reasoning systems. It consists of two main components—the JLambda programming language and interpreter and the Glyphish hierarchy—together with additional classes for organizing window elements.

JLambda is an untyped Scheme-like lexically scoped interpreted language, that provides a runtime interface to available Java classes, using Java's built-in reflective capabilities. Since it is interpreted, JLambda expressions can be evaluated at runtime.

The Glyphish hierarchy is a Java class hierarchy that provides abstractions for creating interactive graphical objects, including specifying their shape,

color, location, and behavior (reaction to events). Groups of glyphish objects can be encapsulated as a single glyphish object. The g2d package also provides classes for defining and displaying graphs whose nodes and edges are glyphish objects.

4 The Maude-NPA GUI in IOP-IMaude

The graphical interface for Maude-NPA that we have developed using IOP-IMaude is publicly available [9], including an installation and user manual and some protocol examples.

There are two key aspects to developing an interactive graphical assistant using IOP-IMaude. Namely, what objects of *the model* (Maude-NPA in our case) to represent (and how), and what requests to the model can be made via these objects. Once those two aspects are decided, there are two parts to implement: extending IMaude to handle those input requests and produce the result to such input requests as descriptions of graphical objects to be presented to the user. Usually, the input requests are sent as IOP messages from the graphics2d actor to the Maude actor (as the result of a Java event in the graphical interface such as clicking a button) and the output to each request is sent as an IOP message from the Maude actor back to the graphics2d actor. Note that the descriptions of graphical objects generated as the output to a request are JLambda functions that realize the descriptions produced by the IMaude NPA Assistant, including other necessary actions such as organizing window elements, menus and toolbars. There is substantial flexibility as to how much detail is filled in by IMaude and how much is done in JLambda; or, in other words, how much of the interface is dynamically generated by IMaude and how much of it is statically created at the design stage. Figure 1 summarizes the Maude-NPA GUI infrastructure.

Let us now explain in detail one concrete sequence of actions and events, namely generating more levels of the Maude-NPA backwards search space. As explained in Section 5 below, this action is activated by clicking the "Next" button. First, at design stage, we created a JLambda function `nextClosure` as the event listener of the "Next" button. When clicked, this JLambda function retrieves the number of levels to be generated and *sequentially* calls another JLambda function `nextClosureRec` as many times as levels to generate. This function is ultimately responsible for sending the request to IMaude:

```
(sinvoke "g2d.util.ActorMsg"
   "send" "maude" gname (concat "nextLevel "  (int 1)))
```

[9] At http://www.dsic.upv.es/grupos/elp/Maude-NPA_GUI

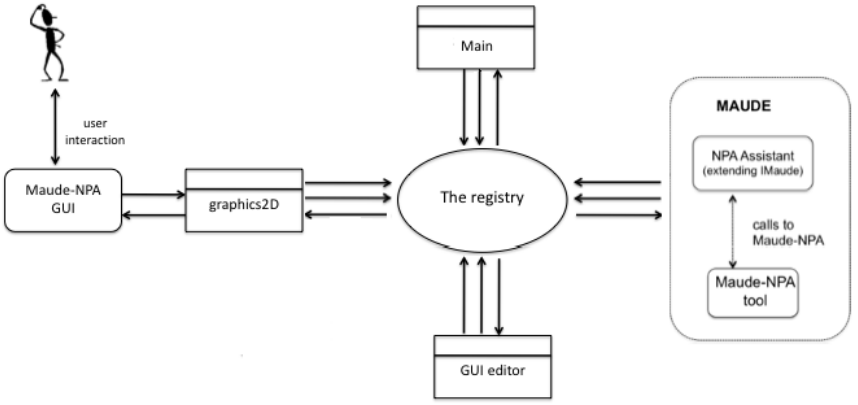

Fig. 1. Overview of the Maude-NPA GUI.

that calls the method `send` of the Glyphish class `ActorMsg` in order to make the graphics2d actor send an IOP message to the Maude actor (denoted by the string `maude`) on behalf of the search tree graph object (denoted by the variable `gname`) such that the contents of the IOP message is a string of the form "`nextLevel 1`".

On the IMaude side, the received IOP message is added to the request queue and is processed using the following (partially described) Maude rewrite rule:

```
crl[nextLevel]:
 [nil,st(processing(req('nextLevel,...)),wait4s,reqQ,es,log),outQ]
=>
 [nil,st(ready,wait4s,(reqQ reqQ1),es',log),outQ]
if es' := nextLevel(...)
/\ reqQ1 := req('extendTree,...,
            req('redisplayNPATree,...,reqQ0)) .
```

where the Maude operator `nextLevel` is ultimately responsible for calling Maude-NPA and generating a new environment `es'` that contains the next level of the search space as a string to be transmitted. In this rewrite rule, two more requests `reqQ1` are added to the IMaude request queue `reqQevel`, namely `extendTree` and `redisplayNPATree`. These two requests will be processed by IMaude rewrite rules as follows. The rule for `extendTree`: (i) extends the IMaude local copy of the search tree, (ii) sends a request `extendNPATree` containing the string stored in `es'` to the graphics2d actor, and (iii) waits for the graphics2d actor to acknowledge. In response, the graphics2d actor, applies the defined JLambda function `extendNPATree` that processes the string, extracting a description of the new tree nodes, adding the information to the search tree graph, and sending an acknowledgement to IMaude. When

IMaude receives the acknowledgement, the request `redisplayNPATree` is processed and a `redisplayNPATreeExp` request is sent to the graphics2d actor, which forces it to redisplay the graphic representation of the search space. The user then sees the extended search tree.

5 A Sample Session with the Interface

The current version of the GUI developed for Maude-NPA allows the user to analyze a crypto protocol specification by displaying the search space tree obtained by Maude-NPA. Apart from displaying the search space as a tree, the user can obtain the textual information of each state in the tree and, as a new special feature, can obtain a pictorial representation of the state strands. Let us consider the famous Needham-Schroeder public key protocol (NSPK) for demonstrating the tool. We recall the informal specification of NSPK, as follows:

$$A \rightarrow B : pk(B, N_A; A)$$

$$B \rightarrow A : pk(A, N_A; N_B)$$

$$A \rightarrow B : pk(B, N_B)$$

where N_A and N_B are nonces generated by the respective principals. The description of the NSPK protocol using strands is as follows:

$$:: r :: [\ pke(B, n(A, r); A)^+,\ pke(A, n(A, r); N_B)^-,\ pke(B, N_B)^+\ ]$$

$$:: r' :: [\ pke(B, N_A; A)^-,\ pke(A, N_A; n(B, r'))^+,\ pke(B, n(B, r'))^-\ ]$$

The first step is to select the protocol to be analyzed. The user can either load his/her own protocol specification file or select one of the provided examples. As explained in Section 2, Maude-NPA must generate the grammars associated to the attack state. The user can provide a grammar file if he/she has analyzed the same protocol specification before, avoiding its generation. Otherwise, Maude-NPA will generate it, thus increasing the load time. More than one attack state can be defined in the protocol specification and the user must select one. After the protocol, the attack state, and (possibly) the grammars have been selected, a new window with the initialized search space tree appears. This initial tree contains a node called "root node", which is the default parent node, and, then, the first level of the search space tree.

One or more levels can be added to the tree by telling Maude-NPA, through the GUI, to do so. There exists a button called "Next" (see Figure 2) which generates a number of levels of the backwards search space. Each node of the tree represents a state of the backwards search. Its background color is

lavender if it is a regular node and *green* if it is an initial state, i.e., a solution node. If a state has no children (predecessor states), its corresponding node at the search space tree is given a *white* color. Figure 2 shows the search space tree for the NSPK protocol when a solution node is found.

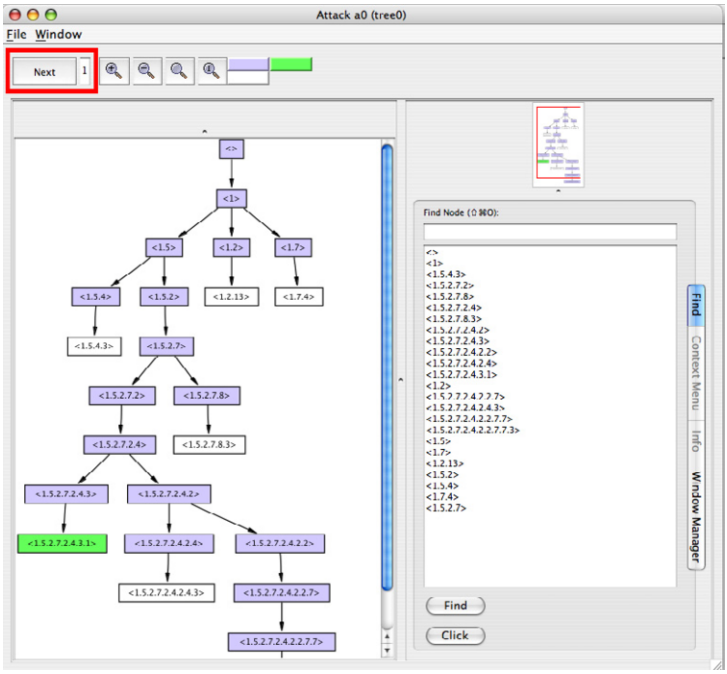

Fig. 2. Search space tree for the NSPK protocol.

As explained in Section 2, for each state, Maude-NPA gives: (i) the current strands, (ii) the intruder knowledge, (iii) the sequence of messages, and (iv) some additional data. There is a contextual menu associated to each node of the search tree which allows the user to, either consult the textual state information given by Maude-NPA, or to obtain a graphical representation of the strands and intruder knowledge information given by Maude-NPA. For the graphical display of strands we followed the original graphical representation of strands shapes [27], but modified to represent the Maude-NPA notion of time. Figure 3 shows the graphical representation of a Maude-NPA state.

A strand is drawn as a vertical sequence of dots connected between them by a double vertical line. Each dot corresponds to an input or output message in the strand and is also called a *node*. As explained in Section 2, both intruder and honest principals behaviors are represented with strands and, thus, shown in the graphical representation. To differentiate between an intruder and an honest strand, we use *grey* and *black* for honest strands and light and dark *green* for intruder strands. The aim of having two colors for each type of strand

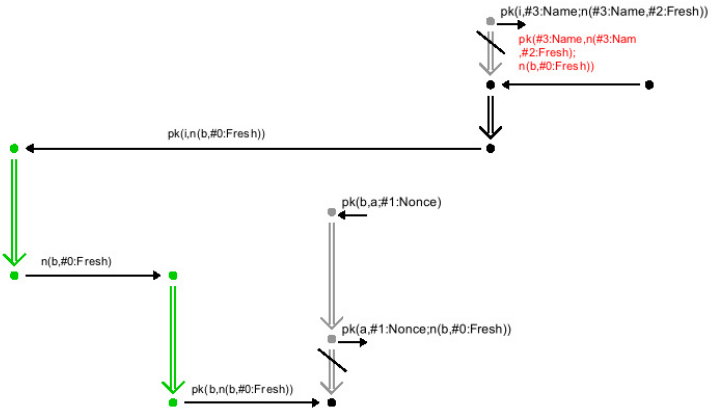

Fig. 3. Strands representation.

is to represent the notion of time: lighter colors for the past and present, and darker colors for the future. The vertical bar used in Maude-NPA for denoting the time position is here represented by a slanted line.

The intruder knowledge ($t \in I$ and $t \notin I$) is integrated into the graphical representation by using different colors for the messages attached to each node. We use *red* if the message is known by the intruder ($t \in I$) and *black* if the message is not known ($t \notin I$). For messages that do not yet belong to a concrete strand, we use a single dot instead of a vertical sequence of dots.

We would like to conclude this section showing how the initial state associated to the NSPK protocol is represented by both its textual and its graphical representations in Figures 4 and 5, respectively.

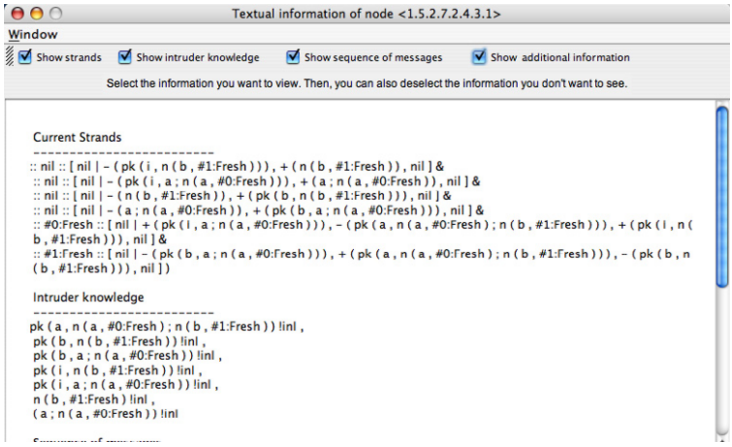

Fig. 4. Textual information of the initial state found for the NSPK protocol.

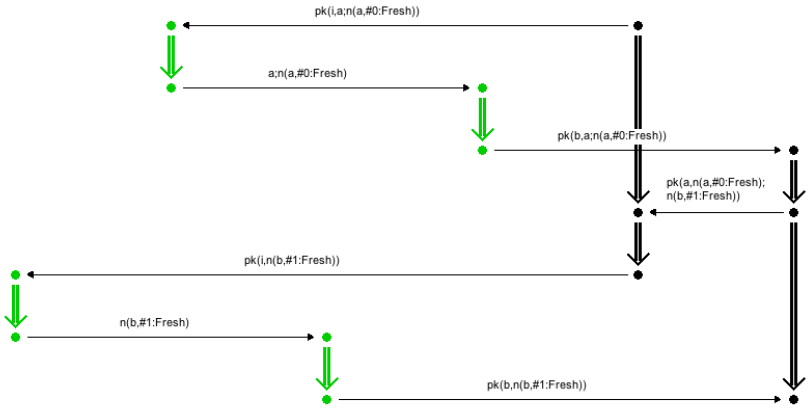

Fig. 5. Graphical display of the initial state found for the NSPK protocol.

6 Conclusions and Future Work

We have presented a graphical user interface for the Maude-NPA protocol analysis tool. Maude-NPA is both a framework for modular reasoning about protocols with different equational theories and an implementation of that framework in Maude, incorporating several of those theories. The Maude-NPA GUI allows the user to visualize the output of Maude-NPA at different levels of detail and to incrementally probe the search space. The strands appearing in the search space can be displayed either as terms or by using a graphical representation that shows the ordering between different protocol messages in a natural way. These features make the analysis of the results produced by Maude-NPA easier to understand and simplify the task of analyzing a crypto protocol.

Future work on the GUI includes improving the graphical visualization of Maude-NPA strands and extending the GUI to allow the user to specify crypto protocols using graphical input, with helpful syntactic checks. A long-term goal is to automate extracting and visualizing information from Maude-NPA analyses in a way that will not only tell the user that his/her protocol is incorrect, but also where the security bug is located and how it could be repaired.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[2] G. Agha. Concurrent Object-oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] M. Anlauff, D. Pavlovic, R. Waldinger, and S. Westfold. Proving Authentication Properties in the Protocol Derivation Assistant. In *Proceedings of Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis*, 2006.

[4] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.C. Heam, O. Kouchnarenko, J. Mantovani, S. Moedersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Vigano, and L. Vigneron. The Avispa Tool for the automated validation of internet security protocols and applications. In *Proceedings of CAV 05*. Springer-Verlag, 2005.

[5] Henry G. Baker and Carl Hewitt. Laws for Communicating Parallel Processes. In Bruce Gilchrist, editor, *IFIP Proceedings of the International Federation for Information Processing Congress 77, Toronto, Canada, August 8-12, 1977*, pages 987–992, August 1977.

[6] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[7] Y. Boichut, T. Genet, Y. Glouche, and O. Heen. Using Animation to Improve Formal Specifications of Security Protocols. In *Proceedings of SAR-SSI 2007, 2nd Conference on Security in Network Architectures and Information Systems, 12-15 June 2007, Annecy, France*, Jun 2007.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] C. Cremers. *Scyther - Semantics and Verification of Security Protocols*. PhD thesis, Eindhoven University of Technology, 2006.

[10] G. Denker and C. L. Talcott. Formal Checklists for Remote Agent Dependability. In *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[11] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties. *Theor. Comput. Sci.*, 367(1-2):162–202, 2006.

[12] Santiago Escobar, Catherine Meadows, and José Meseguer. State Space Reduction in the Maude-NRL Protocol Analyzer. In Sushil Jajodia and Javier López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2008.

[13] Santiago Escobar, Catherine Meadows, and Jose Meseguer. *Maude-NPA, Version 1.0*, March 2009. Available at http://maude.cs.uiuc.edu/tools/Maude-NPA.

[14] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proceedings of 1973 International Joint Conference on Artificial Intelligence*, pages 235–245, August 1973.

[15] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[16] S.B. Freedman J.K. Millen, S.C. Clark. The Interrogator: Protocol Secuity Analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, February 1987.

[17] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. *Software - Concepts and Tools*, 17(3):93–102, 1996.

[18] Ian A. Mason and Carolyn L. Talcott. IOP: The InterOperability Platform & IMaude: An Interactive Extension of Maude. *Electr. Notes Theor. Comput. Sci.*, 117:315–333, 2005.

[19] Catherine Meadows. The NRL Protocol Analyzer: An Overview. *The Journal of Logic Programming*, 26(2):113–131, 1996.

[20] José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[21] José Meseguer. Rewriting Logic and Maude: a Wide-Spectrum Semantic Framework for Object-Based Distributed Systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV, IFIF TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), September 6-8, 2000, Stanford, California, USA*, volume 177 of *IFIP Conference Proceedings*, pages 89–. Kluwer, 2000.

[22] J. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1997.

[23] D. Pavlovic and C. Meadows. Deriving Secrecy in Key Establishment Protocols. In Juris Hartmanis Gerhard Goos and Jan van Leeuwen, editors, *11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006. Proceedings*, volume 4189 of *Lecture Notes in Computer Science*, pages 384–403. Springer-Verlag, 2006.

[24] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-Complete. In *Proceedings of CSFW, 14th IEEE Computer Security Foundations Workshop, 11-13 June 2001, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society, 2001.

[25] Carolyn Talcott. Pathway logic. In *Formal Methods for Computational Systems Biology*, volume 5016 of *LNCS*, pages 21–53. Springer, 2008. 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems.

[26] Carolyn Talcott and David L. Dill. Multiple representations of biological processes. *Transactions on Computational Systems Biology*, 2006.

[27] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7(1), 1999.